

Lecture 12: Access Methods

“If you don’t find it in the index,
look very carefully through the
entire catalog”

- Sears, Roebuck and Co., Consumers Guide, 1897

What you will learn about in this section

1. Indexes: Basics
2. ACTIVITY: Creating indexes

Indexes: High-level

- An index on a file speeds up selections on the search key fields for the index.
 - Search key properties
 - Any subset of fields
 - is not the same as *key of a relation*
- *Example:*

Product(name, maker, price)

On which attributes
would you build
indexes?

More precisely

- An index is a **data structure** mapping of a tuple of search keys to sets of rows in a database table
 - Provides efficient lookup & retrieval by search key value- usually much faster than searching through all the rows of the database table
- An index can store the full rows it points to (*primary index*) or pointers to those rows (*secondary index*)
 - We'll mainly consider secondary indexes

Conceptual Example

What if we want to
return all books
published after 1867?
The above table might
be very expensive to
search over row-by-row...

Russian_Novels

BID	Title	Author	Published	Full_text
001	<i>War and Peace</i>	Tolstoy	1869	...
002	<i>Crime and Punishment</i>	Dostoyevsky	1866	...
003	<i>Anna Karenina</i>	Tolstoy	1877	...

```
SELECT *  
FROM Russian_Novels  
WHERE Published > 1867
```

Composite Keys

11	80
12	10
12	20
13	75

<Age, Sal>

Name	Age	Sal
Bob	12	10
Cal	11	80
Luda	12	20
Tara	13	75

80	11
10	12
20	12
75	13

<Sal, Age>

11
12
12
13

<Age>

80
10
20
75

<Sal>

Equality Query:

Age = 12 and Sal = 90?

Range Query:

Age = 5 and Sal > 5?

Composite keys in
Dictionary Order.

On which attributes can we
do range queries?

<age,sal>
not equal to
<sal,age>

Composite Keys

- Pro:
 - When they work they work well
 - We'll see a good case called "index-only" plans or **covering** indexes.
- Con:
 - Guesses? (time and space)

Covering Indexes

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

We say that an index is **covering** for a specific query if the index contains all the needed attributes—*meaning the query can be answered using the index alone!*

The “needed” attributes are the union of those in the SELECT and WHERE clauses...

Example:

```
SELECT Published, BID  
FROM Russian_Novels  
WHERE Published > 1867
```

Activity-12.ipynb

1. B+ Trees

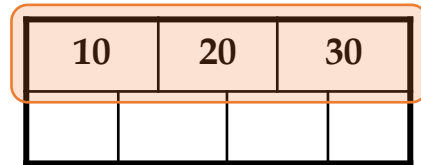
What you will learn about in this section

1. B+ Trees: Basics
2. B+ Trees: Design & Cost
3. Clustered Indexes

B+ Trees

- Search trees
 - B does not mean binary!
- Idea in B Trees:
 - make 1 node = 1 physical page
 - Balanced, height adjusted tree (not the B either)
- Idea in B+ Trees:
 - Make leaves into a linked list (for range queries)

B+ Tree Basics

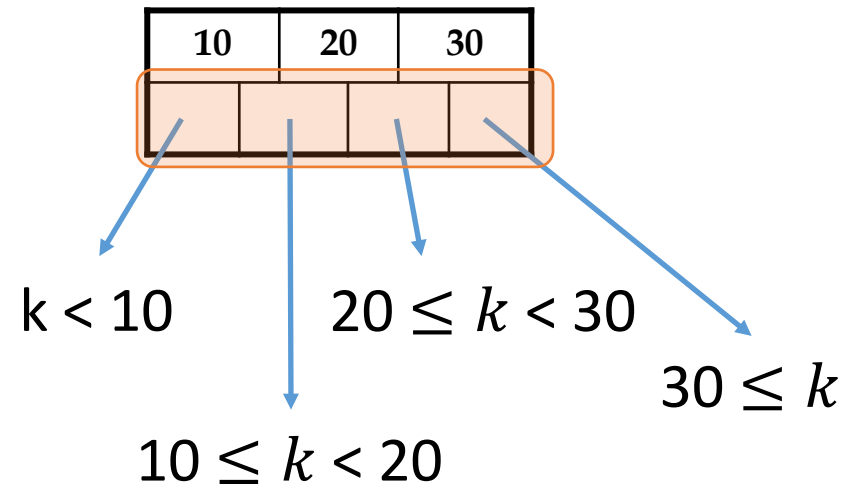


Parameter d = the degree

Each *non-leaf* (“interior”) **node** has $\geq d$ and $\leq 2d$ **keys***

except for root node, which can have between **1 and $2d$ keys*

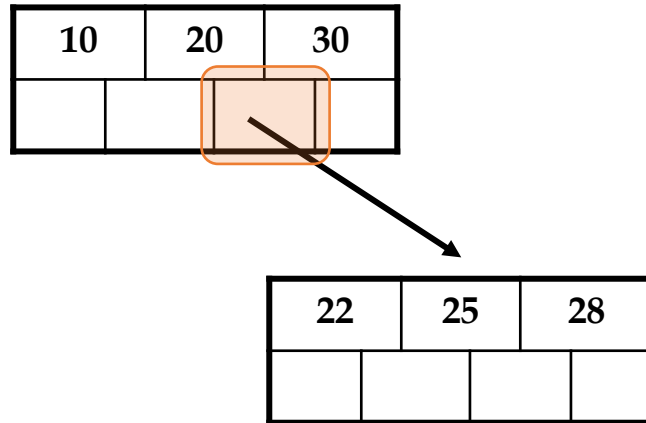
B+ Tree Basics



The n keys in a node define $n+1$ ranges

B+ Tree Basics

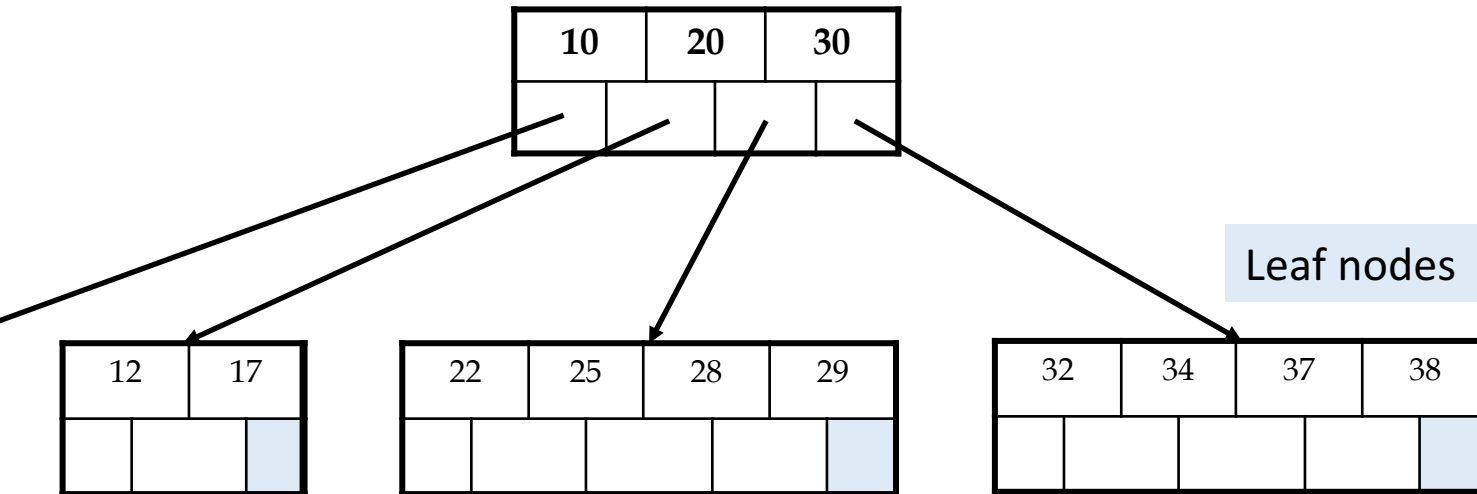
Non-leaf or *internal* node



For each range, in a *non-leaf* node, there is a **pointer** to another node with keys in that range

B+ Tree Basics

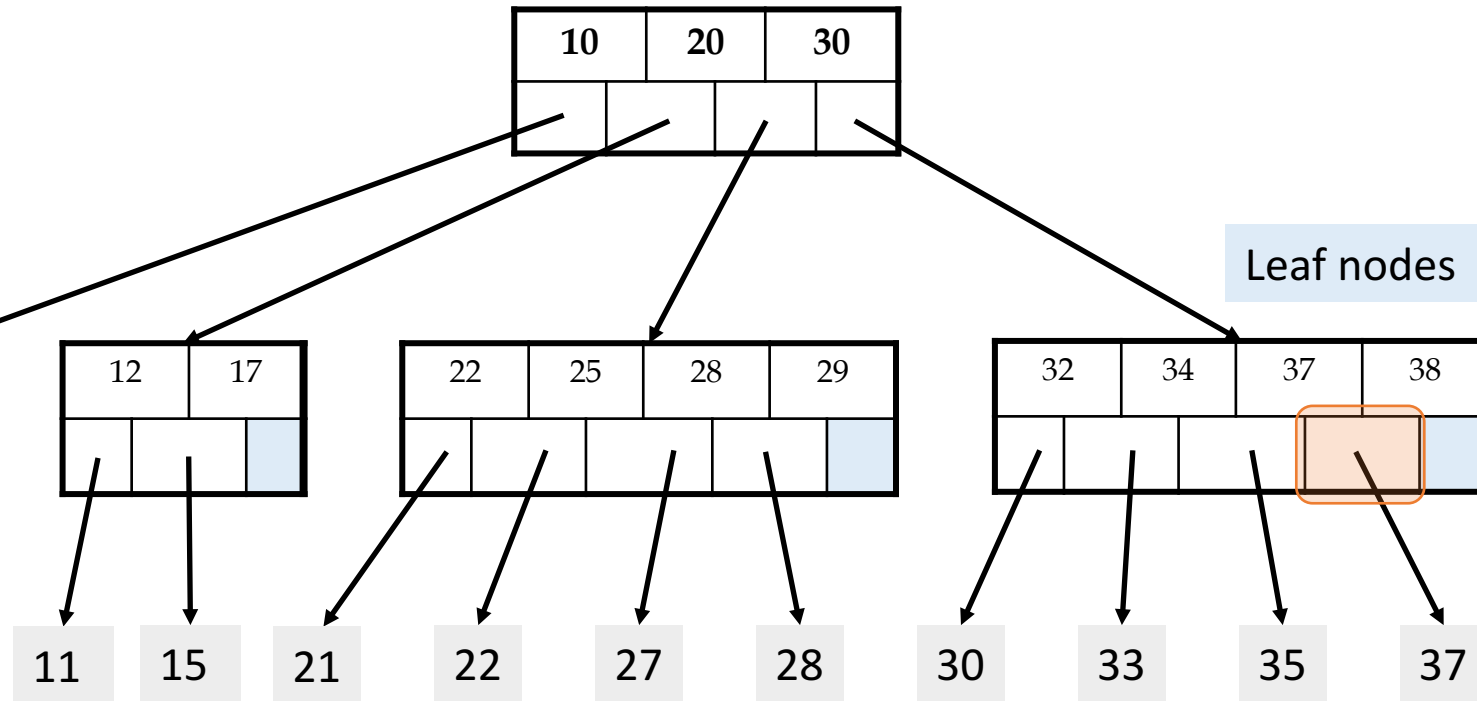
Non-leaf or *internal* node



Leaf nodes also have
between d and $2d$ keys,
and are different in that:

B+ Tree Basics

Non-leaf or *internal* node

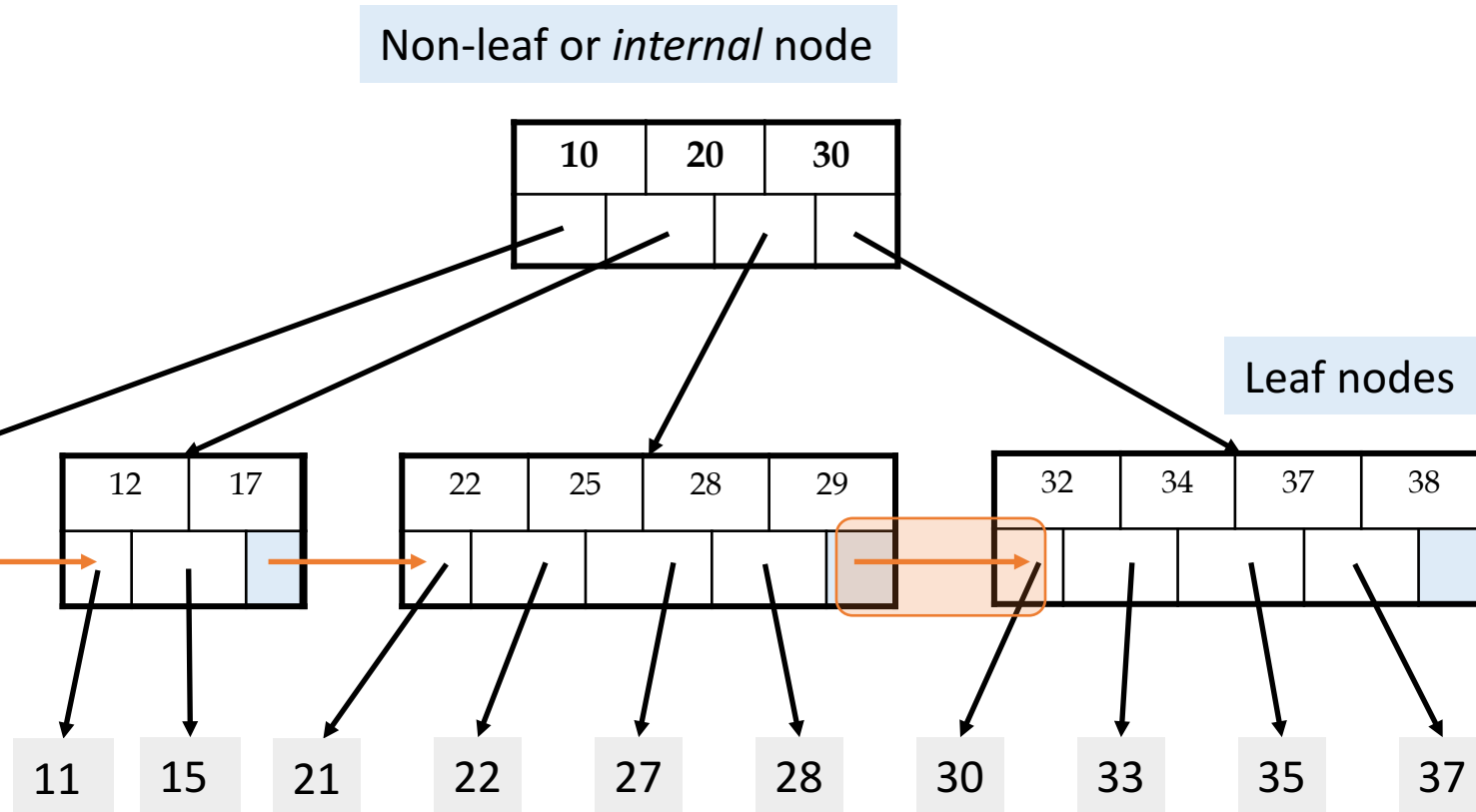


Leaf nodes

Leaf nodes also have between d and $2d$ keys, and are different in that:

Their key slots contain pointers to data records

B+ Tree Basics



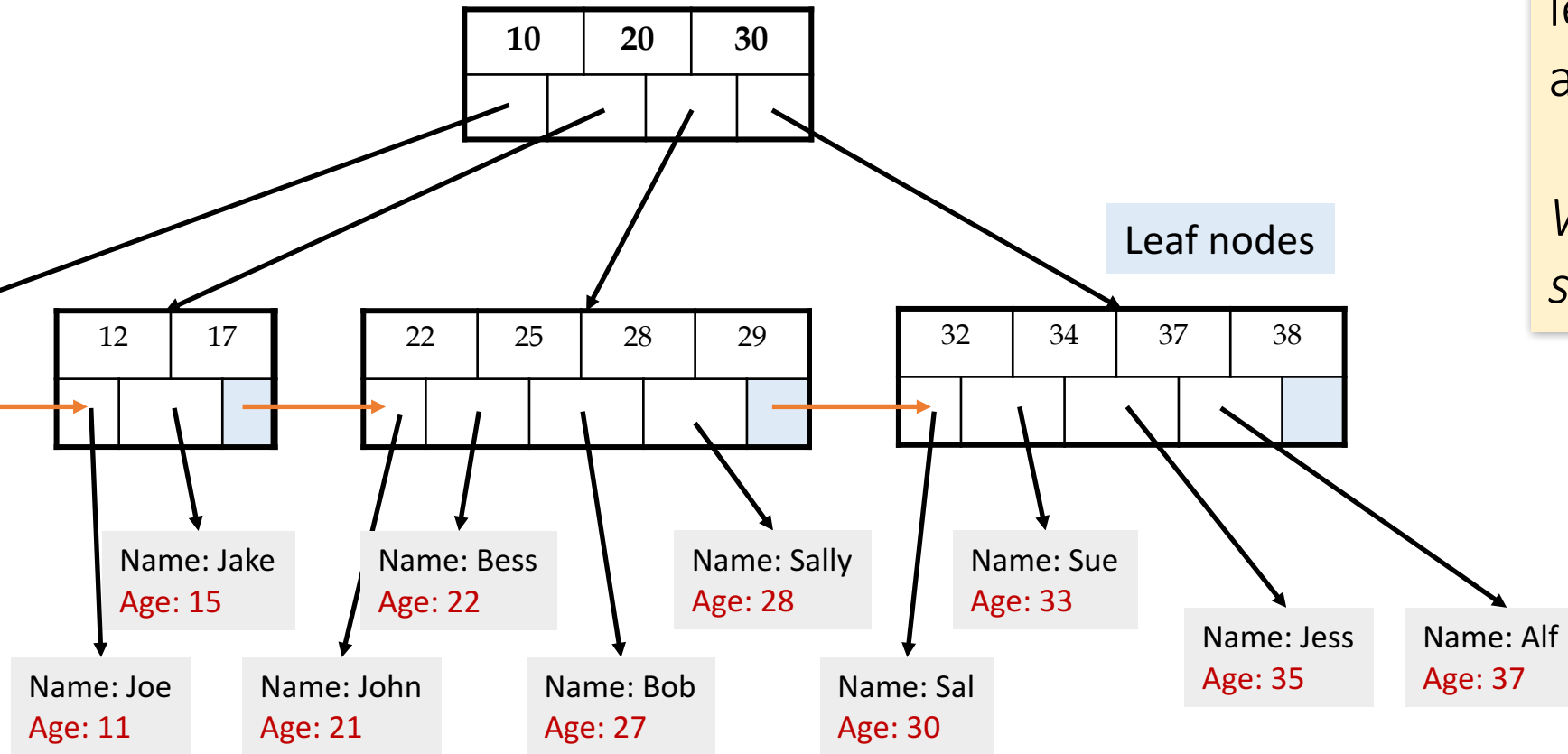
Leaf nodes also have between d and $2d$ keys, and are different in that:

Their key slots contain pointers to data records

They contain a pointer to the next leaf node as well, *for faster sequential traversal*

B+ Tree Basics

Non-leaf or *internal* node



Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display (as before)...

Some finer points of B+ Trees

Searching a B+ Tree

- For exact key values:
 - Start at the root
 - Proceed down, to the leaf
- For range queries:
 - As above
 - *Then sequential traversal*

```
SELECT name  
FROM   people  
WHERE  age = 25
```

```
SELECT name  
FROM   people  
WHERE  20 <= age  
      AND age <= 30
```

B+ Tree Exact Search Animation

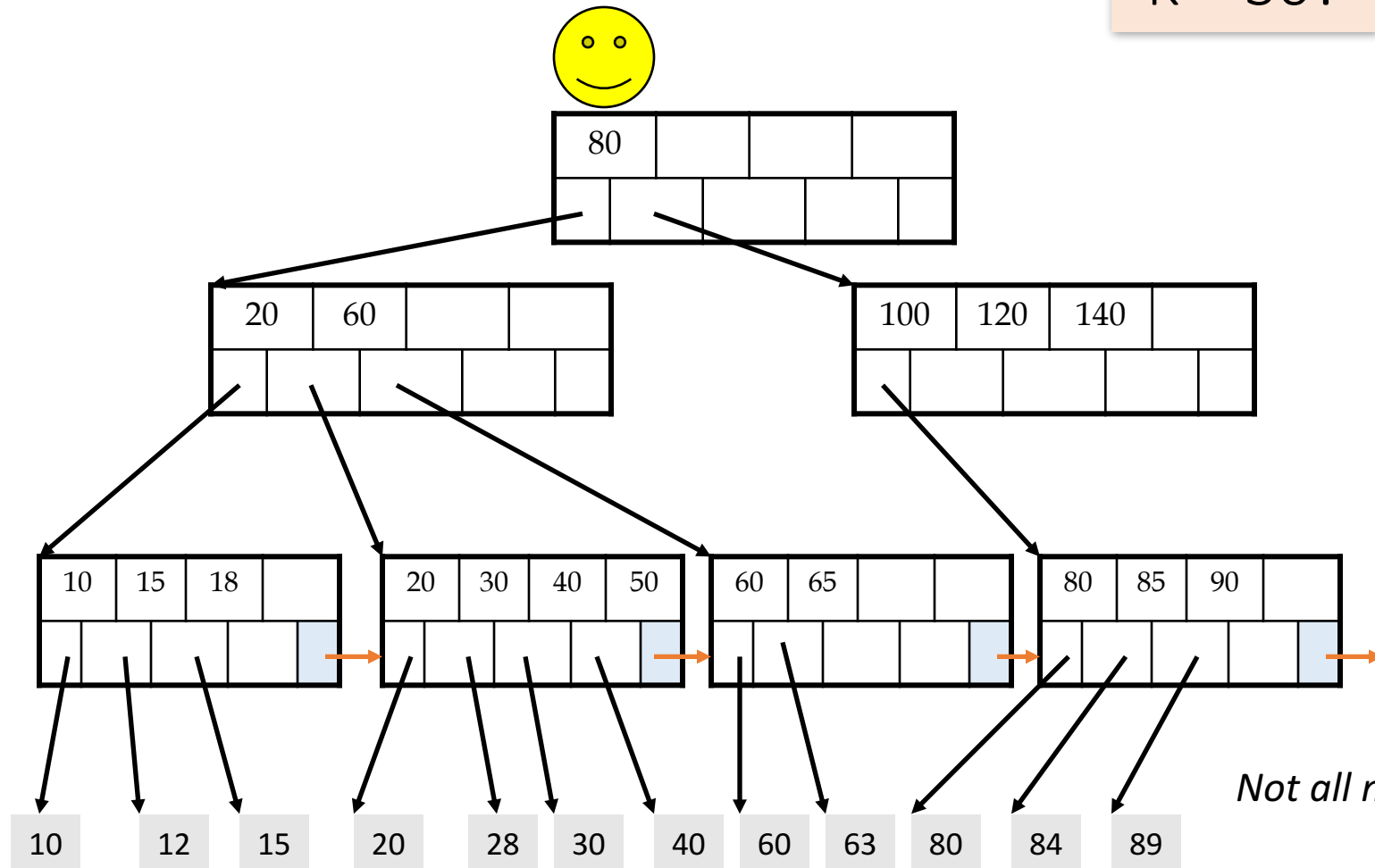
K = 30?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



B+ Tree Range Search Animation

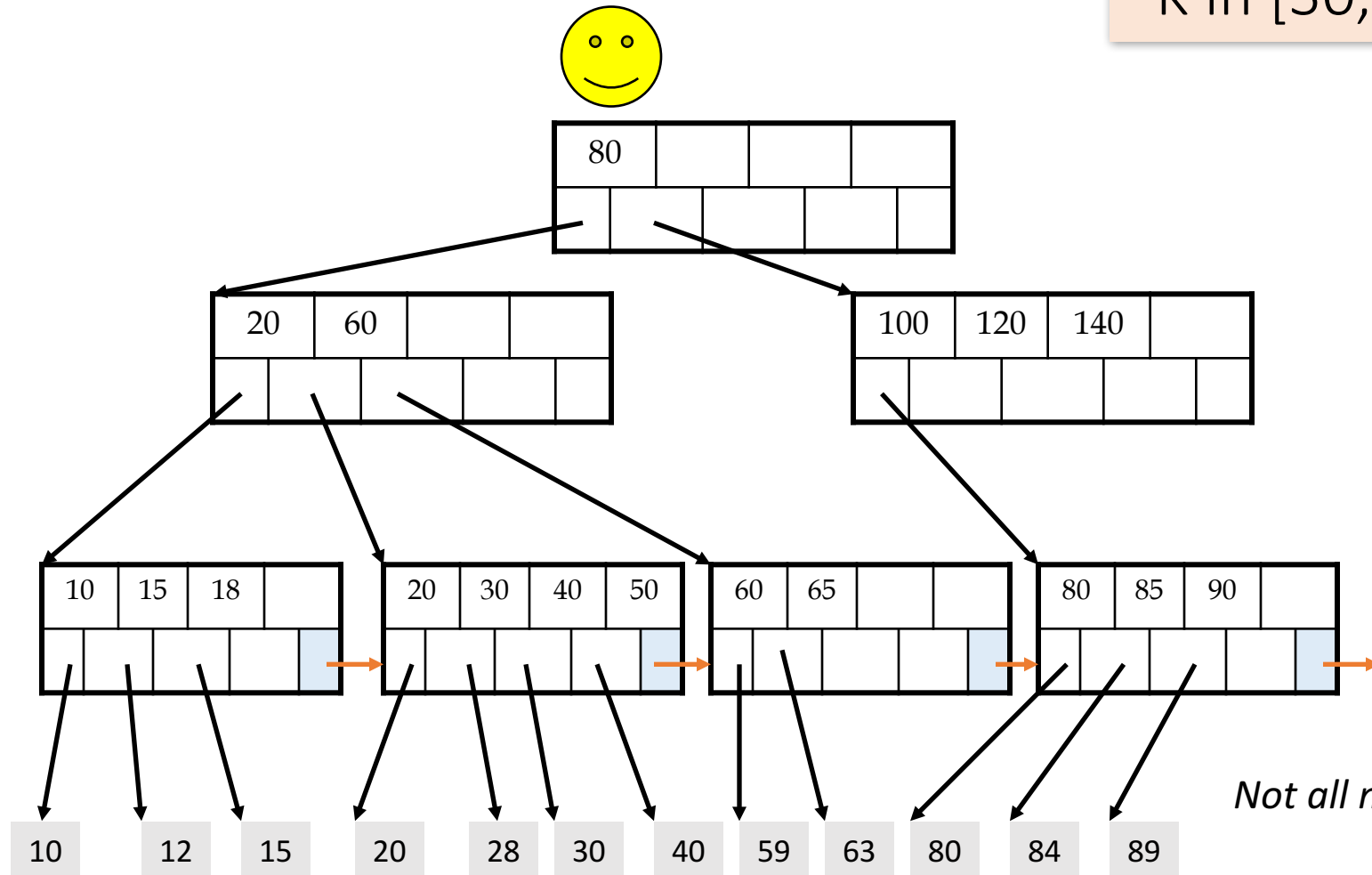
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



B+ Tree Design

- How large is ***d***?
- Example:
 - Key size = 4 bytes
 - Pointer size = 8 bytes
 - Block size = 4096 bytes
- We want each *node* to fit on a single *block/page*
 - $2d \times 4 + (2d+1) \times 8 \leq 4096 \rightarrow d \leq 170$

NB: Oracle allows 64K = 2^{16} byte blocks
 $\rightarrow d \leq 2730$

B+ Tree: High Fanout = Smaller & Lower IO

- As compared to e.g. binary search trees, B+ Trees have **high fanout** (*between $d+1$ and $2d+1$*)
- This means that the **depth of the tree is small**
 - getting to any element requires very few IO operations
 - Most or all of the B+ Tree in main memory!
- A TiB = 2^{40} Bytes. What is the height of a B+ Tree (with fill-factor = 1) that indexes it (with 64K pages)?
 - $(2 * 2730 + 1)^h = 2^{40} \rightarrow h = 4$

The fanout is defined as the number of pointers to child nodes coming out of a node

Fanout *depends on the data* (assume it's constant for simplicity in calculations)

The known universe contains $\sim 10^{80}$ particles... what is the height of a B+ Tree that indexes these?

B+ Trees in Practice

- Typical order: $d=100$. Typical fill-factor: 67%.
 - average fanout = 133
- Typical capacities:
 - Height 4: $133^4 = 312,900,700$ records
 - Height 3: $133^3 = 2,352,637$ records
- Top levels of tree sit *in the buffer pool*:
 - Level 1 = 1 page = 8 Kbytes
 - Level 2 = 133 pages = 1 Mbyte
 - Level 3 = 17,689 pages = 133 MBytes

Fill-factor is the percent of available slots in the B+ Tree that are filled; is usually < 1 to leave slack for (quicker) insertions

What is the relationship between fill factor and fanout?

$$(2d+1)*F = f$$

Typically, only pay for one IO!

Simple Cost Model for Search

- Let:
 - f = fanout, which is in $[d+1, 2d+1]$ (*we'll assume it's constant for our cost model...*)
 - N = the total number of *pages* we need to index
 - F = fill-factor (usually $\sim 2/3$)
- Our B+ Tree needs to have room to index N/F pages!
 - We have the fill factor in order to leave some open slots for faster insertions
- What height (h) does our B+ Tree need to be?
 - $h=1 \rightarrow$ Just the root node- room to index f pages
 - $h=2 \rightarrow$ f leaf nodes- room to index f^2 pages
 - $h=3 \rightarrow$ f^2 leaf nodes- room to index f^3 pages
 - ...
 - $h \rightarrow f^{h-1}$ leaf nodes- room to index f^h pages!

$\rightarrow$ We need a B+ Tree of height $h = \left\lceil \log_f \frac{N}{F} \right\rceil$!

Simple Cost Model for Search

- Note that if we have **B** available buffer pages, by the same logic:
 - We can store L_B levels of the B+ Tree in memory
 - where **L_B is the number of levels such that the sum of all the levels' nodes fit in the buffer:**
 - $B \geq 1 + f + \dots + f^{L_B-1} = \sum_{l=0}^{L_B-1} f^l$
- In summary: to do exact search:
 - We read in one page per level of the tree
 - However, levels that we can fit in buffer are free!
 - Finally we read in the actual record

$$\text{IO Cost: } \left\lceil \log_f \frac{N}{F} \right\rceil - L_B + 1$$

$$\text{where } B \geq \sum_{l=0}^{L_B-1} f^l$$

Simple Cost Model for Search

- To do range search, we just follow the horizontal pointers
- The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each **page** of the results- we phrase this as “Cost(OUT)”.

$$\text{IO Cost: } \left\lceil \log_f \frac{N}{F} \right\rceil - L_B + \text{Cost}(OUT)$$

$$\text{where } B \geq \sum_{l=0}^{L_B-1} f^l$$

Cost(OUT) has one subtle but important twist... let's watch again

B+ Tree Range Search Animation

K in [30,85]?

How many IOs did our friend do?

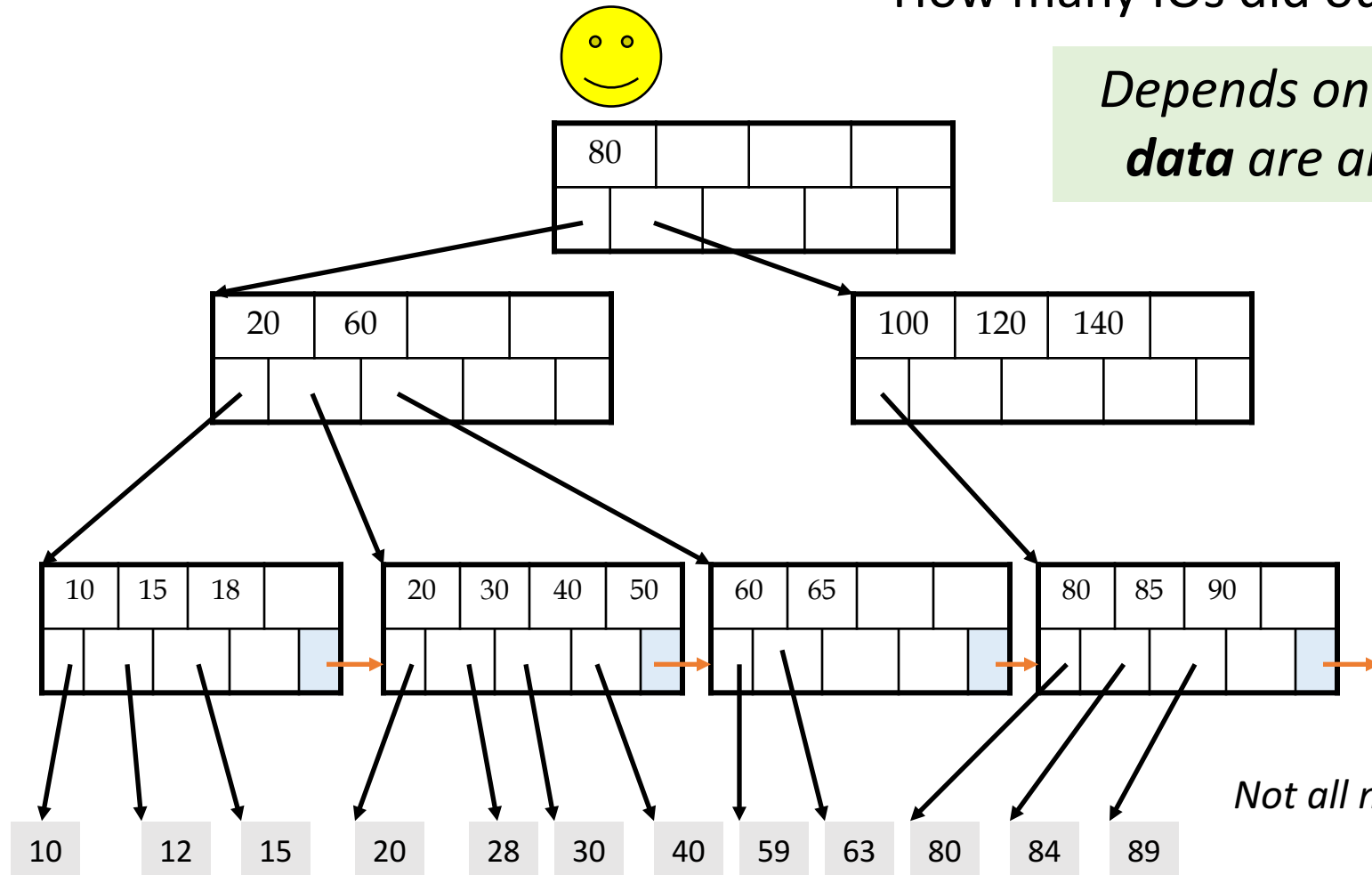
Depends on **how the data** are arranged

30 < 80

30 in [20,60)

30 in [30,40)

To the data!

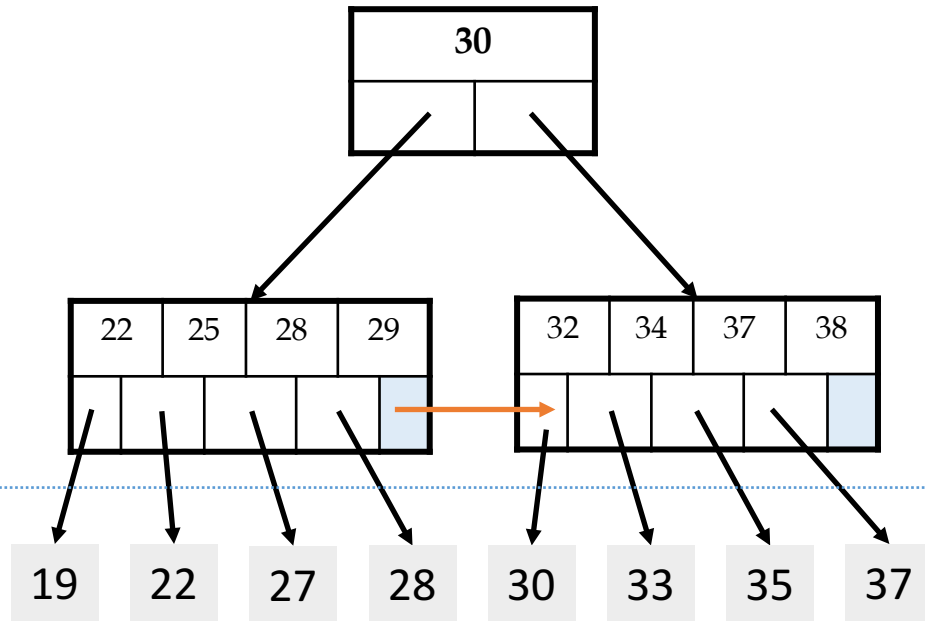


Not all nodes pictured

Clustered Indexes

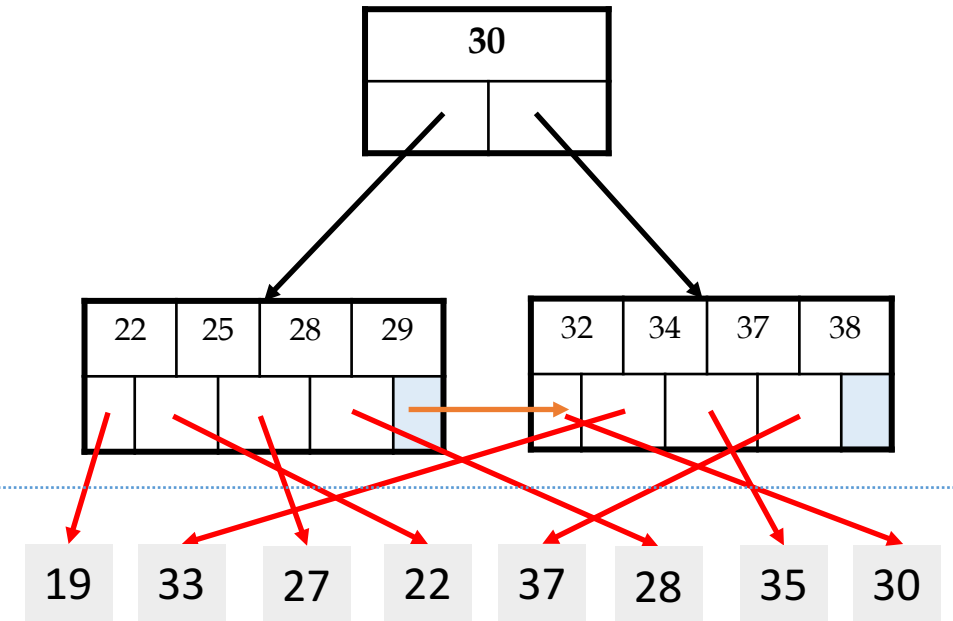
An index is **clustered** if the underlying data is ordered in the same way as the index's data entries.

Clustered vs. Unclustered Index



Clustered

Index Entries



Unclustered

Data Records

Clustered vs. Unclustered Index

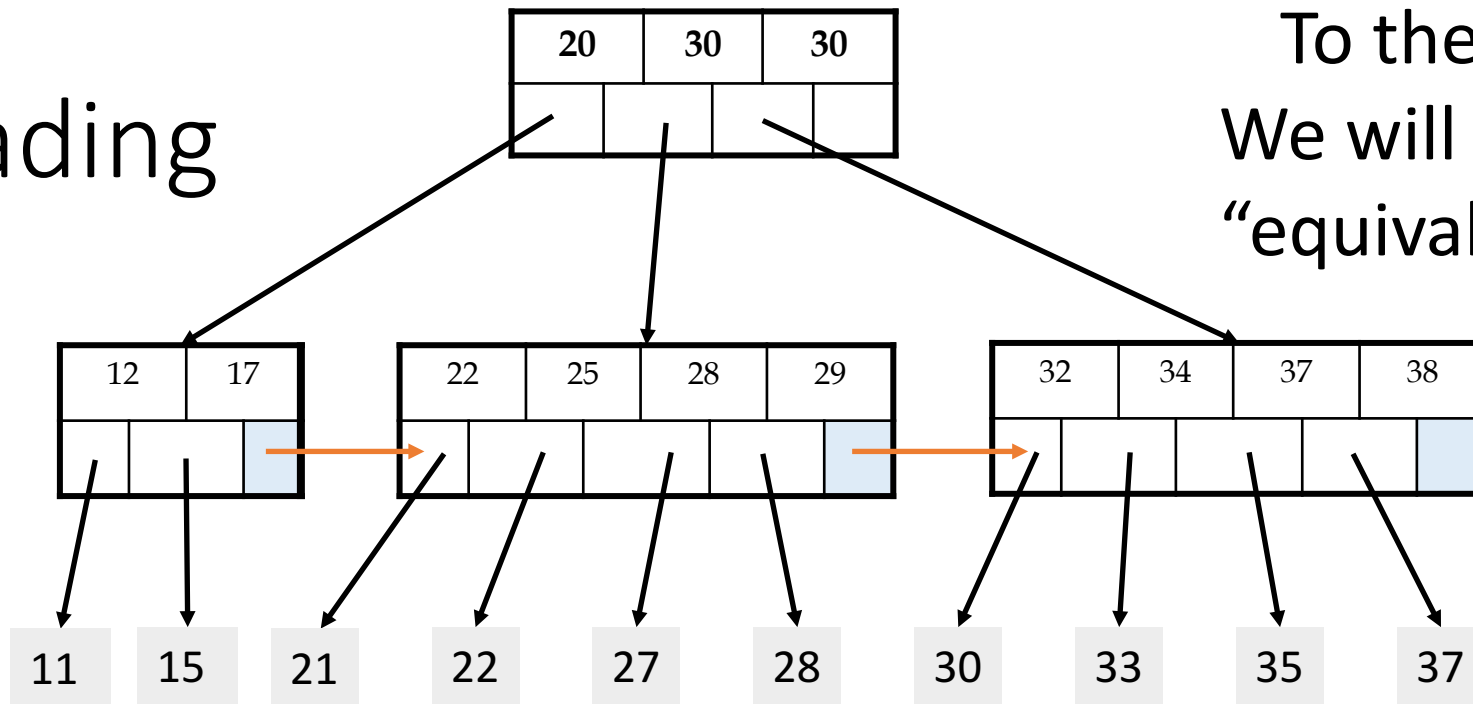
- Recall that for a disk with block access, **sequential IO is much faster than random IO**
- For exact search, no difference between clustered / unclustered
- For range search over R values: difference between **1 random IO + R sequential IO**, and **R random IO**:
 - A random IO costs ~ 10ms (sequential much much faster)
 - For R = 100,000 records- **difference between ~10ms and ~17min!**

Fast Insertions & Self-Balancing

- We won't go into specifics of B+ Tree insertion algorithm, but has several attractive qualities:
 - ~ Same cost as exact search
 - **Self-balancing:** B+ Tree remains **balanced** (with respect to height) even after insert

B+ Trees also (relatively) fast for single insertions!
However, can become bottleneck if many insertions (if fill-factor slack is used up...)

Bulk Loading



Input: Sorted File
Output: B+ Tree

Message: Bulk Loading is faster!

Summary

- We covered an algorithm + some optimizations for sorting larger-than-memory files efficiently
 - An ***IO aware*** algorithm!
- We create **indexes** over tables in order to support ***fast (exact and range) search*** and ***insertion*** over ***multiple search keys***
- **B+ Trees** are one index data structure which support very fast exact and range search & insertion via ***high fanout***
 - ***Clustered vs. unclustered*** makes a big difference for range queries too

2. Nested Loop Joins

What you will learn about in this section

1. RECAP: Joins
2. Nested Loop Join (NLJ)
3. Block Nested Loop Join (BNLJ)
4. Index Nested Loop Join (INLJ)

RECAP: Joins

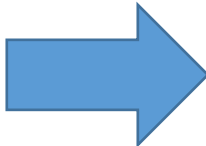
Joins: Example

 $R \bowtie S$

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

R			S					
A	B	C	A	D				
1	0	1	3	7				
2	3	4	2	2				
2	5	2	2	3				
3	1	1						



A	B	C	D
2	3	4	2

Joins: Example

 $R \bowtie S$

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

R			S					
A	B	C	A	D				
1	0	1	3	7				
2	3	4	2	2				
2	5	2	2	3				
3	1	1			A	B	C	D
					2	3	4	2
					2	3	4	3

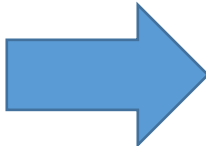
Joins: Example

 $R \bowtie S$

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

R			S					
A	B	C	A	D				
1	0	1	3	7				
2	3	4	2	2				
2	5	2	2	3				
3	1	1						



A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2

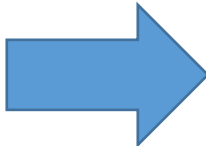
Joins: Example

 $R \bowtie S$

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

R			S					
A	B	C	A	D				
1	0	1	3	7				
2	3	4	2	2				
2	5	2	2	3				
3	1	1						



A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3

Joins: Example

 $R \bowtie S$

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

S

A	D
3	7
2	2
2	3

→

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Semantically: A Subset of the Cross Product

 $R \bowtie S$

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples $r \in R, s \in S$ such that $r.A = s.A$

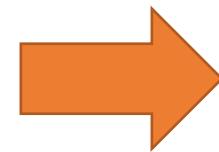
R

A	B	C
1	0	1
2	3	4
2	5	2
3	1	1

×

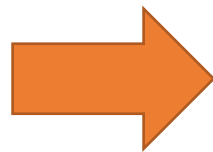
S

A	D
3	7
2	2
2	3



Cross
Product

...



Filter by
conditions
($r.A = s.A$)

A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

Can we actually
implement a join
in this way?

Notes

- We write $R \bowtie S$ to mean *join R and S by returning all tuple pairs where **all shared attributes** are equal*
- We write $R \bowtie S \text{ on } A$ to mean *join R and S by returning all tuple pairs where **attribute(s) A** are equal*
- For simplicity, we'll consider joins on **two tables** and with **equality constraints** (“equijoins”)

However joins *can* merge > 2 tables, and some algorithms do support non-equality constraints!

Nested Loop Joins

Notes

- We are again considering “IO aware” algorithms:
care about disk IO
- Given a relation R , let:
 - $T(R)$ = # of tuples in R
 - $P(R)$ = # of pages in R
- Note also that we omit ceilings in calculations...
good exercise to put back in!

Recall that we read / write
entire pages with disk IO

Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

Nested Loop Join (NLJ)

Compute $R \bowtie S$ on A :

```
for r in R:
```

```
    for s in S:
```

```
        if r[A] == s[A]:
```

```
            yield (r,s)
```

Cost:

$P(R)$

1. Loop over the tuples in R

Note that our IO cost is based on the number of *pages* loaded, not the number of tuples!

Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

Cost:

$$P(R) + T(R) * P(S)$$

1. Loop over the tuples in R
2. For every tuple in R , loop over all the tuples in S

Have to read *all of S* from disk for *every tuple in R !*

Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :
```

```
  for  $r$  in  $R$ :
```

```
    for  $s$  in  $S$ :
```

```
      if  $r[A] == s[A]$ :
```

```
        yield  $(r, s)$ 
```

Cost:

$$P(R) + T(R) * P(S)$$

1. Loop over the tuples in R
2. For every tuple in R , loop over all the tuples in S
3. **Check against join conditions**

Note that NLJ can handle things other than equality constraints... just check in the *if* statement!

Nested Loop Join (NLJ)

```

Compute  $R \bowtie S$  on  $A$ :
  for  $r$  in  $R$ :
    for  $s$  in  $S$ :
      if  $r[A] == s[A]$ :
        yield ( $r, s$ )
  
```

What would *OUT* be if our join condition is trivial (if *TRUE*)?

OUT could be bigger than $P(R)*P(S)$... but usually not that bad

Cost:

$$P(R) + T(R)*P(S) + OUT$$

1. Loop over the tuples in R
2. For every tuple in R , loop over all the tuples in S
3. Check against join conditions
4. **Write out (to page, then when page full, to disk)**

Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

Cost:

$$P(R) + T(R) * P(S) + \text{OUT}$$

What if R ("outer") and S ("inner") switched?



$$P(S) + T(S) * P(R) + \text{OUT}$$

Outer vs. inner selection makes a huge difference-
DBMS needs to know which relation is smaller!

IO-Aware Approach

Block Nested Loop Join (BNLJ)

Given $B+1$ pages of memory

Cost:

Compute $R \bowtie S$ on A :

for each $B-1$ pages pr of R :

for page ps of S :

for each tuple r in pr :

for each tuple s in ps :

if $r[A] == s[A]$:

yield (r,s)

$P(R)$

1. Load in $B-1$ pages of R at a time (leaving 1 page each free for S & output)

Note: There could be some speedup here due to the fact that we're reading in multiple pages sequentially however we'll ignore this here!

Block Nested Loop Join (BNLJ)

Given $B+1$ pages of memory

Cost:

Compute $R \bowtie S$ on A :

for each $B-1$ pages pr of R :

for page ps of S :

for each tuple r in pr :

for each tuple s in ps :

if $r[A] == s[A]$:

yield (r,s)

$$P(R) + \frac{P(R)}{B-1} P(S)$$

1. Load in $B-1$ pages of R at a time (leaving 1 page each free for S & output)

2. For each $(B-1)$ -page segment of R , load each page of S

Note: Faster to iterate over the *smaller* relation first!

Block Nested Loop Join (BNLJ)

Given $B+1$ pages of memory

Cost:

Compute $R \bowtie S$ on A :

```

for each B-1 pages pr of R:
  for page ps of S:
    for each tuple r in pr:
      for each tuple s in ps:
        if r[A] == s[A]:
          yield (r,s)
  
```

$$P(R) + \frac{P(R)}{B-1} P(S)$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. Check against the join conditions

BNLJ can also handle non-equality constraints

Block Nested Loop Join (BNLJ)

Given **$B+1$** pages of memory

Cost:

Compute $R \bowtie S$ on A :

```

for each B-1 pages pr of R:
  for page ps of S:
    for each tuple r in pr:
      for each tuple s in ps:
        if r[A] == s[A]:
          yield (r,s)
  
```

$$P(R) + \frac{P(R)}{B-1} P(S) + OUT$$

1. Load in B-1 pages of R at a time (leaving 1 page each free for S & output)
2. For each (B-1)-page segment of R, load each page of S
3. Check against the join conditions

4. Write out

Again, ***OUT*** could be bigger than $P(R)*P(S)$... but usually not that bad

BNLJ vs. NLJ: Benefits of IO Aware

- In BNLJ, by loading larger chunks of R, we minimize the number of full *disk reads* of S
 - We only read all of S from disk for ***every (B-1)-page segment of R!***
 - Still the full cross-product, but more done only *in memory*

NLJ

$$P(R) + T(R) * P(S) + \text{OUT}$$



BNLJ

$$P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT}$$

BNLJ is faster by roughly $\frac{(B-1)T(R)}{P(R)}$!

BNLJ vs. NLJ: Benefits of IO Aware

- Example:
 - R: 500 pages
 - S: 1000 pages
 - 100 tuples / page
 - We have 12 pages of memory ($B = 11$)
- NLJ: Cost = $500 + 50,000 * 1000 = 50 \text{ Million IOs} \approx \underline{140 \text{ hours}}$
- BNLJ: Cost = $500 + \frac{500 * 1000}{10} = 50 \text{ Thousand IOs} \approx \underline{0.14 \text{ hours}}$

Ignoring OUT here...

A very real difference from a small
change in the algorithm!

Smarter than Cross-Products

Smarter than Cross-Products: From Quadratic to Nearly Linear

- All joins that compute the **full cross-product** have some **quadratic** term

- For example we saw:

$$\text{NLJ} \quad P(R) + \mathbf{T(R)P(S)} + \text{OUT}$$

$$\text{BNLJ} \quad P(R) + \frac{P(R)}{B-1} \mathbf{P(S)} + \text{OUT}$$

- Now we'll see some (nearly) linear joins:
 - $\sim O(P(R) + P(S) + \mathbf{OUT})$, where again **OUT** could be quadratic but is usually better

We get this gain by *taking advantage of structure*- moving to equality constraints (“equijoin”) only!

Index Nested Loop Join (INLJ)

Compute $R \bowtie S$ on A :

Given index idx on $S.A$:

for r in R :

s in $idx(r[A])$:

 yield r, s

Cost:

$$P(R) + T(R) * L + OUT$$

where L is the IO cost to access all the distinct values in the index; assuming these fit on one page, $L \sim 3$ is good est.

→ We can use an **index** (e.g. B+ Tree) to *avoid doing the full cross-product!*

Summary

- We covered joins--an ***IO aware*** algorithm makes a big difference.
- Fundamental strategies: blocking and reorder loops (asymmetric costs in IO)
- Comparing nested loop join cost calculation is something that I will definitely ask you!